COBB

Aug/Sept 1991
Vol. 1 No. 1

INSIDE ASSEMBLER™

Tips & techniques for 80x86 Assembler

Your Premier Issue Is Inside...

WELCOME!

WELCOME!

COBB

INSIDE ASSEMBLER™

Dear Assembly Language Programmer:

It's my pleasure to welcome you to *Inside Assembler*, our bi-monthly journal for assembly language programmers. The Cobb Group is delighted to have you with us, and we look forward to serving you over the coming months.

By subscribing to this one-of-a-kind journal, you've taken a significant step toward having--right at your fingertips--proven techniques and timesaving shortcuts to help you program your computer more efficiently.

As publisher of *Inside Assembler*, my goal is to make this publication one of the most useful on your reference shelf. I encourage you to send your comments, criticisms, suggestions, and ideas for articles you'd like to see published.

Enjoy your premier issue of *Inside Assembler*.

Sincerely,

Douglas Cobb
Publisher

P.S. For a preview copy of upcoming topics, see the back cover.

COBB

Aug/Sep 1991
Vol. 1 No. 1

INSIDE ASSEMBLER

Tips & techniques for Assembler • PC

Examining the Program Segment Prefix (PSP)

By Marco C. Mason

Listing 1 starts on page 13

DOS is full of fascinating bits and pieces. One very interesting data structure contained in DOS is the Program Segment Prefix (PSP). In this article, you'll get an overview of the structure of the PSP, then we'll present SHOWPSP.ASM, a program that displays some of the PSP's fields.

What the PSP looks like

The PSP is a 256-byte (100h) data structure that contains a table of useful information. Some of this information is for use by your program, and some is strictly for DOS' own use. Table A shows some of the data items in the PSP.

Table A: *Description of the PSP*

Offset Address	Size (bytes)	Description
00H	2	Abort program
02H	2	Last segment available
05H	5	DOS service dispatch
0AH	4	Terminate address
0EH	4	^Break exit address
12H	4	Critical error exit address
16H	2	PSP of parent
18H	20	File handle table
2CH	2	Environment segment
2EH	4	Stack storage for INT 21H
32H	2	File handle table size
34H	4	File handle table address
38H	4	Pointer to previous PSP
50H	3	DOS service dispatch
5CH	36	Default FCB #1
6CH	36	Default FCB #2
80H	1	# characters in command tail
81H	127	Command tail
80H	128	Disk Transfer Area

There are several fields in the PSP that we've not been able to find any documentation for, so we left them out of Table A. We may discuss those fields in a future issue. It's worth mentioning that the entire PSP table comes from an early operating system called CP/M, which MS-DOS was originally modeled after. As a consequence of this, you'll see references to CP/M in the following descriptions. Now let's discuss each of the fields in the PSP that we list in Table A.

00H—Abort program

This field is a vestige of the CP/M era—in CP/M, you could call this address to terminate a program. In deference to this, Microsoft placed an INT 20H instruction at location 00H in the PSP. Since INT 20H also performs the Abort program service, you can still terminate a program by calling address 00H in the PSP.

Note, however, that this is discouraged. The *best* way to terminate a program is to call DOS service 4CH, which cleans up after your program and allows you to pass an errorlevel code back to DOS. Note also that Microsoft Assembler Version 6.0's `.EXIT` directive uses DOS service 4CH, so it's no more difficult to use.

02H—Last segment available

When DOS starts your program, it allocates the largest piece of RAM available to your program. After it does so, it puts the segment address of the segment *following* the last available segment into address 02H in the PSP. Therefore, if you have a 640K machine with nothing loaded at the end of memory, this field contains 0A000H, which

Continued on page 2

IN THIS ISSUE

INSIDE ASSEMBLER

Inside Assembler (ISSN 1057-560X) is published bi-monthly by The Cobb Group.

Prices	Domestic ..... $59/yr. ($14 each) Outside US ..... $79/yr. ($17 each)
Address	The Cobb Group 9420 Bunsen Parkway, Suite 300 Louisville, KY 40220
Phone	Toll-free ..... (800) 223-8720 (502) 491-1900 FAX ..... (502) 491-4200
Staff	Editor-in-Chief ..... Marco C. Mason Publications Mgr ..... Toni Bowers Editing ..... Duane Spurlock Gregory L. Harris Production ..... Gina Sledge Eric Paul Design ..... Karl Feige Publisher ..... Douglas Cobb

Address correspondence and special requests to The Editor, *Inside Assembler*, at the address above. Address subscriptions, fulfillment questions and requests for bulk orders to Customer Relations, at the address above.

Postmaster: Send address changes to *Inside Assembler*, P.O. Box 35160, Louisville, KY 40232. Second class postage is pending in Louisville, KY.

Conventions

When we describe programs, we'll either print them as a figure in the article (if the listing is small) or put the listing at the end of the journal and use callouts when we describe sections of the program. A callout isn't guaranteed to compile or run, because it is only a fragment of code. Complete listings in figures or at the end of the journal will compile and run.

You'll also notice that the journal is peppered with words or phrases in a `monospaced font`. We use this font for directives, instructions, labels, macros, and subroutine names. Register names will appear as normal text. Whenever we reference a directive or instruction in the text, it will appear in uppercase, even though it may not be uppercase in the code.

Another practice you'll see in the journal is that some function names will have a tail consisting of the $ symbol and a sequence of letters. These letters indicate the types of parameters the function accepts. See Table A for the type names associated with each letter.

Table A: *Type names for letters in function name tail*

b - byte (unsigned)	c - character (signed)
d - double word	f - far pointer
i - integer (signed)	n - near pointer
s - string (address)	w - word (unsigned)
y - nybble (4 bits)	

Continued from page 1

Examining the PSP

indicates that your program can use all memory from the segment containing the PSP to segment 9FFFH.

If you subtract this value from the segment address of the PSP, you'll get the amount of memory, in paragraphs, available to your program.

05H—DOS service dispatch

This field is another holdover from the CP/M era. In CP/M, you load the A register with the function code and call location 05H to perform a service call. When DOS creates the PSP, it puts a long call to the DOS service dispatcher, so you can still call 05H to invoke DOS. You shouldn't use this method, however, because Microsoft declared it obsolete, and it's much easier to just invoke INT 21H.

Welcome to *Inside Assembler*!

Complexity—most people program in high-level languages to avoid it. Unfortunately, you pay for it with both speed and code size. In the old days, when both RAM and speed were expensive, assembly language ruled the kingdom.

These days RAM and computer speed are much cheaper. It would almost seem that the need to program in assembly language doesn't exist anymore. This reasoning doesn't apply, however, because today's programs are much more complex and they *require* as much RAM and CPU speed as they can get.

So as much as ever, we have the tradeoff between speed, size, and complexity.

In this, the Cobb Group's newest journal, we'll present some of the techniques that professionals use when they write assembly language programs. We'll present macros and subroutines to make your programming easier, as well as articles describing how to organize your code and debug it more efficiently. For the beginners among you, we'll describe the inner workings of the assembler routines and macros, so you can build your assembly language programming skills.

Our first issue is, of course, a guess about what you want. It's your responsibility to correspond with us here at the Cobb Group, so we can adjust the journal to fit your needs. Remember, we're the best because we care about giving you what you want.

In future issues, we'll dedicate this letters column to your questions, comments, tips, bug discoveries, and techniques. We welcome your comments, criticisms, and questions, so don't be shy!

Marco C. Mason,
Editor-in-Chief

0AH—Terminate address

When your program ends, it has to go somewhere. DOS looks at location 0AH to find the address of the code to execute when your program completes. Initially, DOS sets it to return to the current command shell, but you can set it to execute other functions.

While you *can* execute other functions, you must be careful. By the time DOS executes your code, it has already closed your files and freed the RAM that holds your program. Avoid taking advantage of this feature.

0EH—^Break exit address, and 12H—Critical error exit address

When you or one of your programs spawns a new process, DOS initializes the fields 0EH and 12H with the addresses found in Interrupt vectors 23H and 24H. If your program intercepts either Int 23H or Int 24H, you can restore the original interrupt vector from this stored value.

16H—PSP of parent

In DOS, programs may run other programs. For instance, COMMAND.COM, the usual command shell, is just another program. Whenever a program starts a new program, DOS places the segment address of the original program's PSP in field 16H of the new program's PSP. This way, a program can determine what program started it up.

Unfortunately, COMMAND.COM zeroes out this field or puts its own PSP segment there, so you can't use this field to find out what program started COMMAND.COM.

18H—File handle table

DOS normally allows your programs to have up to 20 files open simultaneously. When you open a file, your program gets a file handle, which is an index into this file handle table. Each file handle contains 0FFH if the file is closed, or DOS' internal file number if it's open.

Starting with DOS 3.3, you can change the number of files you can have open simultaneously, with DOS service 67H. (We'll discuss this further in the section where we discuss fields 32H and 34H.)

2CH—Environment segment

One of DOS' many enhancements, starting in Version 2.0, is the concept of the environment. Your programs can use the environment as a set of variables for holding configuration parameters.

Each time you start a program from the command line, DOS copies all the environment variables into a block of RAM and puts the segment address of the environment into location 2CH in the PSP. DOS terminates each string with a 0 (i.e., it uses the ASCIIZ format for storing them). It terminates the list of environment strings with a zero-length string. (We'll cover the environment in much more detail in a future article.)

2EH—Stack storage for INT 21H

Whenever you perform a DOS service, DOS may use one of its own internal stacks. Since DOS needs to store the current stack pointer somewhere, it stores it as a `DWORD` at address 2EH in the PSP.

32H—File handle table size, and 34H—File handle table address

Starting with DOS 3.3, you can give a program more than 20 file handles. To do this, you use DOS service 67H, which sets up a new file handle table for you. DOS stores the number of available file handles as a `WORD` in address 32H in the PSP, and the address of the current file handle table as a `DWORD` in location 34H of the PSP.

38H—Pointer to previous PSP

Supposedly, starting with DOS 3.0, the 38H field points to the previous PSP. Usually, however, this field contains 0FFFF:0FFFFH, which means it isn't pointing to anything at all.

50H—DOS service dispatch

Another way you can perform a DOS service is with a far call to address 50H in the PSP. (This works only in DOS 3.0 and later.) However, as mentioned for field 05H, it's probably wiser to simply use INT 21H.

5CH—Default FCB #1

When you start a program from the command line, DOS checks the argument list. If you have any arguments, DOS uses the first two to initialize the two default File Control Blocks (FCBs). The first FCB starts at 5CH in the PSP. If you open a file based on the the FCB at 5CH, it overlaps the FCB at 6CH, because each FCB is 36 bytes long. DOS initializes most of the fields in the FCB to zero, and fills in the filename from the first argument in the command line.

6CH—Default FCB #2

As we mentioned in the previous section, DOS initializes the first two FCBs from the first two arguments in the command line. Since each FCB is 36 bytes long, you'll have to move the FCB at 6CH if you want to use the one at 5CH, because the one at 5CH overlaps the one at 6CH.

80H, 81H—Command tail

The fields 80H and 81H in the PSP provide you with the command tail—the text typed in after the program name,

including the carriage return. You can use this data to examine what arguments the user typed.

For example, if you invoke a program with the following line

```
psp fred martha ethel
```

field 80H contains 12H, and field 81H contains `fred martha ethel` and a carriage return.

80H—Disk Transfer Area

The second use of the field starting at 80H is the default Disk Transfer Area (DTA). Since the DTA is 128 bytes long, it fills out the entire address range from 80H to FFH, overlapping both fields 80H and 81H—which store the initial command tail.

The example program: SHOWPSP

The program SHOWPSP.ASM, shown in Listing 1 on page 13, prints the useful information in the PSP. Most of the program is easy to follow once you understand the two macros we use heavily in it.

Figure A shows the definition of the macro `printStr`, which we use to print strings in either the DS or ES segment.

Figure A: *The printStr macro*

```
printStr macro offs:REQ, segreg

    IFNB <segreg>
        push    ds
        mov     dx, segreg
        mov     ds, dx
    ENDIF
    mov     dx, offset offs
    DOSsvc  9
    IFNB <segreg>
        pop     ds
    ENDIF

endm
```

Let's look at what `printStr` actually does. First it checks to see if you specified a segment with the `IFNB` directive. If you did, `printStr` pushes the DS register (so the macro can retrieve it later), and loads the value `segreg` into DS.

Next the macro loads the DX register with `offs`, the address of the string to print. Then it invokes DOS service 9, which prints a string pointed to by DS:DX. (Note that DOS expects a `$` at the end of the string—this is how DOS knows when to quit printing characters.) Finally, if you specified a segment, `printStr` must retrieve the old DS value it put on the stack for safekeeping.

That's all there is to it: We simply tell `printStr` what string we want to print, and if the string is not in the default data segment, we specify the segment to use.

The other important macro we use is `printLine`, which prints a string followed by a hexadecimal number. Figure B shows the definition of the `printLine` macro.

Figure B: *The printLine macro*

```
printLine macro string, type, address

    mov     dx, offset string
    DOSsvc  9
    IF type EQ BYTE
        mov al, es:[address]
        call hexout$b
    ELSEIF type EQ WORD
        mov ax, es:[address]
        call hexout$w
    ELSEIF type EQ DWORD
        mov ax, es:[address]
        mov dx, es:[address+2]
        call hexout$d
    ENDIF

endm
```

The `printLine` macro first puts the address of a string into DX and prints it with DOS service 9. Then it checks what type of value to print. If you tell `printLine` to print a `BYTE`, it loads a byte into the AL register and calls the `hexout$b` function to print it. Similarly, if you request `printLine` to print a `WORD`, it loads the word into the AX register and calls `hexout$w` to print it. When you direct `printLine` to print a `DWORD`, it loads the most significant word into DX and the least significant word into AX, and calls `hexout$d` to print the double word with a colon between the words.

Now let's look at what SHOWPSP prints on a typical machine. Figure C shows the screen after you run SHOWPSP with the following command line:

```
showpsp martha clyde ethel
```

Figure C: *Typical SHOWPSP results*

```
[C:\COBB\IASM\WORK\DOSDATA]showpsp martha clyde ethel

        Current PSP segment is - 1768
      Last segment of PSP seg  - 9FF4
            Terminate address - 0762:01DC
          ^Break exit address   - 0762:014B
   Critical error exit address - 0762:0156
                Parent's PSP - 0762
         Segment of environment - 175C
Stack contents at last INT 21H - 1797:0648
          Number of file handles - 0014
           Address of file table - 1768:0018
         Pointer to parent's PSP - FFFF:FFFF
                  Default file 1 - MARTHA
                  Default file 2 - CLYDE
Characters in command line tail - 13
Command tail:  martha clyde ethel
File handle table:
      01 01 01 00 02 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
[C:\COBB\IASM\WORK\DOSDATA]
```

Notice that the PSP segment address is at 01768H, and the last segment address is at 09FF4H. This means that SHOWPSP has 0888CH paragraphs (or 559,296 bytes)

What do YOU want to see in *Inside Assembler?*

Please rate your interest in the following types of articles by circling a number for each, with 5 indicating the types of articles you would most like to see in ***Inside Assembler***.

Tutorial articles on assembly language programming 1 2 3 4 5
Discussions on interfacing assembly language functions with other languages 1 2 3 4 5
Articles presenting working functions and macros 1 2 3 4 5
Articles on using the Programmer's WorkBench and other programming tools 1 2 3 4 5
Reviews of third-party tools compatible with MASM 1 2 3 4 5
Reviews and demonstrations of third-party code libraries 1 2 3 4 5

In articles that present fully functional programs, in what detail would you like us to explain the code ?

Little detail Moderate detail Great detail

Are there any specific topics you would like to see covered in upcoming issues?

_____

_____

_____

_____

Please complete this card and drop it in a mailbox. No postage is necessary.

QUICK SURVEY

NO POSTAGE
NECESSARY
IF MAILED IN
THE UNITED
STATES

BUSINESS REPLY MAIL
FIRST CLASS MAIL PERMIT NO. 618 LOUISVILLE, KENTUCKY

POSTAGE WILL BE PAID BY ADDRESSEE

INSIDE ASSEMBLER
EDITOR-IN-CHIEF
PO BOX 35160
LOUISVILLE KY 40232-9719

available to it. Since it's not a TSR and doesn't consume any memory, the next program you load will have about 559K available to it.

Another interesting feature is that the address of the file table field points to the default location, address 18H in the PSP. When DOS starts a program, it automatically opens five handles for it. You'll notice that the first three file handles all hold the same value: 1, which stands for the console (the console is the standard input device, the standard output device, and the error output device). The auxiliary device (fourth file handle) holds 0, which represents the COM port; and the printer device (fifth file handle) holds 2, which represents the printer port.

Conclusion

Now you have a basic understanding of DOS' oldest data structure—the PSP. In it, DOS stores some interesting bits of information that you can take advantage of in your programs. Additionally, the PSP has some ancient remnants of CP/M conventions that are better left alone.

USEFUL OUTPUT ROUTINES

A collection of hexadecimal output functions

Listing 2 starts on page 14

To be useful, a program needs to do one important thing—output data in a useful form. For this reason, programmers are always concerned about I/O. Since many of our programs in this and future issues are going to print hexadecimal numbers to the screen, we decided that our first step should be to create a small set of functions that format and output hexadecimal numbers. You'll find our functions in Listing 2 on page 14.

In this article, we'll discuss two related sets of functions—some formatting functions and some output functions. The output functions use the formatting functions, so let's cover the formatting functions first.

hexfmt$yn—format a nybble

The basis of all our hexadecimal routines is the `hexfmt$yn` function, which formats a nybble. You pass it a nybble in AL and a pointer to the output buffer in DI (as the `yn` at the end of it indicates—see the conventions on page 2 for more information), and it places the formatted character into the buffer and adjusts the pointer to the next location in the buffer. Since understanding this function is so critical to understanding the rest of the hexadecimal output library, let's examine it in detail. Figure A contains the code for `hexfmt$yn`.

Figure A: *The function hexfmt$yn*

```
hexfmt$yn proc

    and     al, 0fh       ; Discard the upper nybble
    add     al, '0'       ; Shift range from 0..f to '0'..'?'
    cmp     al, '9'       ; If in range of ':'..'?' shift the
    jng     @F            ; range to 'A'..'F'
    add     al, 'A'-':'
@@: mov     [di], al
    inc     di
    ret

hexfmt$yn endp
```

First, the function masks off the upper nybble by ANDing AL with 0fh. This keeps only the lower four bits in the AL register. Next, `hexfmt$yn` adds the ASCII equivalent of 0 to the AL register, which shifts the range of characters to '0' through '?'. The first nine characters in the range correspond to 0 through 9, and the next six are :, ;, <, =, >, and ?. Since we really want these last six characters to be A..F, the function needs to add the difference between : and A to the digit *only if* we don't have a numeric value. Therefore, the function compares the character in AL with 9, and if the character is larger than 9, the function adjusts the value to the range of characters A..F. Next, `hexfmt$yn` puts the character into the buffer and increments the buffer pointer, DI, and returns.

hexfmt$bn—format a byte

Now that you see how the `hexfmt$yn` function works, understanding `hexfmt$bn` is easy. You pass the byte to format in the AL register, and `hexfmt$bn` puts the formatted characters into the address pointed to by the DI register.

First `hexfmt$bn` saves a copy of the AX register on the stack (to preserve the lower nybble in AL), then it shifts the upper nybble into the lower nybble position. Since there is now a nybble to format in the lower four bits of AL, and a pointer to the buffer in DI, we can use the previous function `hexfmt$yn` to format the character. That's how we format the upper nybble.

Formatting the lower nybble is just as simple: We restore our copy of AX from the stack, and call `hexfmt$yn` again. Now we've formatted both the upper *and* lower nybbles of AL into the buffer pointed to by DI.

hexfmt$wn—format a word

The `hexfmt$wn` function is just as simple as the `hexfmt$bn` function—and it operates almost identically. It accepts a

word in the AX register and formats the word into the buffer pointed to by the DI register.

To do so, it swaps the upper and lower bytes in the AX register and calls `hexfmt$bn` to format the upper byte. Then it swaps the AH and AL registers again and calls `hexfmt$bn` to format the lower byte. That's all there is to it.

hexfmt$dn—format a double word

The `hexfmt$dn` function has just two slight quirks in it. First it accepts the double word as a pair of word registers: DX and AX. While we could have used the EAX register to hold the double word, this would prevent us from using our routine on 8088 through 80286 CPUs. (The EAX register is the 32 bit version of AX for the 80386 and later processors.) As usual, you pass the address of the buffer to hold the formatted text in the DI register.

`hexfmt$dn` operates nearly identically to `hexfmt$wn`: It first swaps the upper and lower words to print, then calls `hexfmt$wn` to format the upper word. Then comes the second quirk—it moves a : into the buffer. We fashioned `hexfmt$dn` to do this because that's what DEBUG and Codeview do when they display double words. Then the function swaps the upper and lower words back to their original positions and calls `hexfmt$wn` to format the lower word.

The hexadecimal output functions

While formatting the text in a string is often useful, you'll probably more often want to print the values directly to the console. The `hexout` functions perform this job. Let's take a look at these output functions.

hexout$y—output a nybble to the screen

Just as `hexfmt$yn` was the key to understanding all the other `hexfmt` functions, `hexout$y` is the key to understanding the rest of the `hexout` functions. Figure B shows the code for `hexout$y`.

First `hexout$y` preserves the DI and DX registers on the stack so `hexout$y` can restore them after it destroys their current contents. Next, it loads DX with the address of `hexbuff`—a scratch buffer included in the hexadecimal output library. Then the function copies this address into DI, so `hexfmt$yn` knows where to put the formatted characters. After `hexout$y` calls `hexfmt$yn` to format the nybble and put it into the buffer, `hexout$y` adds a `$` to the end of the buffer to terminate the string. Then it invokes DOS service 9 to print the string pointed to by the DX register—this prints the nybble onto the screen.

Figure B: *The hexout$y function*

```
hexout$y proc

    @SaveRegs di, dx            ; Preserve DI & DX
    mov     dx, offset hexbuff  ; Start of hex output buff
    mov     di, dx              ; Set ptr for hexfmt$yn
    call    hexfmt$yn           ; Format char into buffer
    mov     byte ptr [di],'$'   ; Terminate the string
    DOSsvc 9                    ; Print the string
    @RestoreRegs                ; Restore DI & DX
    ret

hexout$y endp
```

hexout$b, hexout$w, hexout$d—output a byte, word, or double word to the screen

The functions `hexout$b`, `hexout$w`, and `hexout$d` all work the same as `hexout$y`. The only difference is that `hexout$b` calls `hexfmt$bn` to format a byte into the buffer, `hexout$w` calls `hexfmt$wn` to format a word into the buffer, and `hexout$d` calls `hexfmt$dn` for double words.

Conclusion

You'll want to get to know these hexadecimal output functions because we'll use them heavily in issues to come. We've even used them in two programs in this issue: SHOWPSP (from the article "Examining the Program Segment Prefix PSP") and TESTMSBN (from the article "Translating between GWBASIC and MASM floating point number formats"). In our next issue, we'll cover an equally important aspect of programming—input.

INTERNAL DATA FORMAT

How Microsoft Assembler represents floating point numbers

Usually, when you write assembly language programs, you don't care how the assembler actually stores floating point numbers. Sometimes, however, that information can come in very handy. In this article, we'll discuss in detail how the Microsoft Macro Assembler stores floating point numbers in memory. Since Microsoft uses the same format used by the 80*x*87 coprocessor, you'll find the information very useful.

Floating point numbers

Nearly all computers and compilers use the same general form for storing floating point numbers. There are three major parts to a floating point number: The exponent, the mantissa, and the sign bit. The exponent tells the computer where the floating point is located, the mantissa holds the significant digits, and the sign bit indicates whether the number is positive or negative.

We humans typically use the same format when a number gets too large or too small to represent it conveniently in a fixed-point representation. For instance, the numbers 38 and 3.14159 are conveniently expressible as fixed-point numbers, while 0.00000000000000000406 and 3,890,000,000,000,000,000,000 aren't. We would typically represent the latter two numbers as $4.06 * 10^{-20}$ and $3.89 * 10^{21}$. For numbers like $3.89 * 10^{21}$, we call 3.89 the mantissa and 21 the exponent.

Because computer representations typically are of fixed size, both the exponent and mantissa are limited in magnitude. This is where some of the differences between computerized representations of floating point numbers appear. The methods used to represent the exponent and mantissa also provide some differences.

The exponent

Since computers use binary instead of decimal digits, the exponent doesn't tell you how many decimal digits to move the decimal point. Instead, it tells you the number of binary digits (bits) to move the decimal point (or perhaps we should say, binary point). So instead of multiplying the mantissa by 10 to the exponent, you multiply the mantissa by 2 to the exponent.

The exponent has to encompass a wide range to be useful. It must cover both large and small numbers. Typically, computers implement the exponent as a biased unsigned value. This means the exponent ranges from 0 to the maximum value, with 0 representing the lowest exponent and the maximum value representing the highest exponent. You compute the actual value of the exponent by subtracting the bias from the value held in the exponent field.

For example, if you have an exponent that you represent in eight bits and bias it with 0x7f, you can represent the range of exponents from -127 (00H-7FH) to 128 (0FFH-7FH). If you want to represent 2^{30}, you simply add 0x7f to 30 to get 0x9d. Conversely, if the exponent field holds 0x51, the exponent represents 0x51-0x7f, which comes to -0x2e or -46, so the exponent represented by 0x51 is 2^{-46} (about $1.421 * 10^{-14}$).

The mantissa and sign

Again, computers use binary instead of decimal digits. So rather than using a decimal fraction to represent a number, the computer uses a binary fraction. Just as humans usually put one significant digit before the decimal point, many computers use the floating point just to the right of the first significant digit. Therefore, the most significant bit of the mantissa is the 2^0 bit (i.e., 1), the next 2^{-1} ($1/2$), etc. We also need a bit to represent the sign of the mantissa. Typically, this sign bit is 1 for negative numbers and 0 for positive ones.

Therefore, if you had a mantissa of eight bits holding the pattern 00101101, and a sign bit of 1, then it represents the fraction - 0.3515625, as shown in Figure A.

Figure A: *Mantissa example*

Bit		Decimal Fraction		Total
0	*	1.0	=	0.0
0	*	0.5	=	0.0
1	*	0.25	=	0.25
0	*	0.125	=	0.0
1	*	0.0625	=	0.0625
1	*	0.03125	=	0.03125
0	*	0.015625	=	0.0
1	*	0.0078125	=	0.0078125
				0.3515625 * -1.0 = - 0.3515625

To get an extra bit of mantissa for free, you can use an interesting trick—normalization, which usually refers to the process of putting something into a standard form. In this case, normalization refers to the act of ensuring that the most significant bit of the mantissa is 1. You can do this by shifting the mantissa left and subtracting 1 from the exponent repeatedly, until the most significant bit is 1.

The reason this buys you an extra bit is that since you know that the most significant bit is 1, you don't need to store it—you already know its value. This trick gives you an extra bit of precision for the entire range of numbers that you can express, but at a cost: How do you represent 0.0? Most computers represent 0.0 with all zeros—the sign, exponent, and mantissa all being 0.

Now you know the basics of floating point number representation on computers. Microsoft Assembler generates code for 80*x*86 and 80*x*87 processors, but the 80*x*87 does all the floating point work. So let's now look at the standard floating-point formats for the 80*x*87.

The 80*x*87 coprocessor

There is a special option available for most IBM PCs and clones—the 80*x*87 math coprocessor. This option is a specialized microprocessor optimized to perform floating-point math operations very quickly. Because of its availability, most computer languages that run on IBM PCs use a data representation compatible with the 80*x*87.

When Intel designed the 80*x*87 series of math coprocessors, it followed IEEE Floating Point Standard 754. This puts a further restriction on the exponent—the

highest value indicates that the value either is Not A Number (NAN) or is an INFinity (INF). The coprocessor uses the NAN value to indicate values that aren't numeric and INF to represent infinite values.

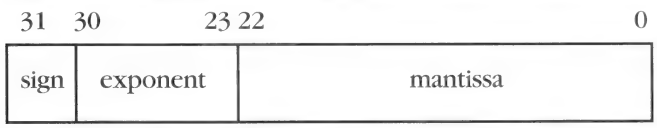
The way the 80*x*87 tells the difference between an INF and a NAN is by the mantissa. If the mantissa is 0, the number is infinite; if the mantissa is nonzero, it's not a number (NAN). When the number is infinite, you can use the sign bit to tell whether the number is a positively or negatively infinite value.

The 80*x*87 has three types of floating point numbers: 32-bit floats, 64-bit floats, and 80-bit floats, also called short, long, and temporary reals, respectively. (It also supports 16-, 32-, and 64-bit integers and 18-digit BCD numbers, but that's a topic for another issue.)

short real—32-bit floating point

The short real takes only four bytes. You can enter a short real in MASM by using the `REAL4` or `DD` directives. The layout of the short real is shown in Figure B.

Figure B: *The short real*



As you can see, the 80*x*87 uses one bit for the sign, eight bits to hold the exponent, and 23 bits to hold the mantissa. The sign bit contains 1 when the mantissa is negative, and 0 if it's positive. The most significant bit in the number is the sign bit, followed by the eight bits of exponent. The 23 bits of mantissa follow.

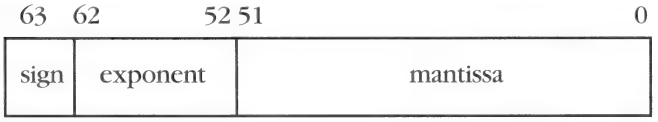
The 80*x*87 (and hence, MASM) biases the exponent at 0x7f, and the mantissa is normalized. Therefore, the exponent can range from 2^{-127} to 2^{127} (remember, IEEE reserved 2128 for NAN and INF). The mantissa can range from 0x800000 (0.5) to 0xFFFFFF (0.99999994). This gives you a range from $\pm 1.7 \times 10^{38}$ to $\pm 1.17 \times 10^{-38}$, with an approximate accuracy of six decimal digits.

long real—64-bit floating point

The long real is quite similar to the short real. Instead of taking four bytes, it takes eight. The sign bit is still the most significant bit, and the exponent still follows the sign bit. However, the exponent extends from eight bits to 11 bits, which increases its range of values. The bias, also, is different—from 0x7f to 0x3ff. The remaining 52 bits represent the mantissa. Figure C shows a long real's layout.

With the extended exponent and mantissa, you can express significantly larger and more precise values. The mantissa range of 52 bits allows you to have approximately 15 decimal digits of accuracy, while the numbers can range from $\pm 2.25 \times 10^{-308}$ to $\pm 1.79 \times 10^{308}$.

Figure C: *The long real*

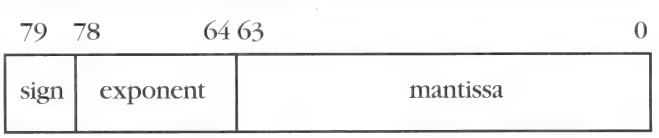


temporary real—80-bit float

The temporary real is slightly different from the short and long reals. The layout is similar, except that the exponent and mantissa are both larger. The difference is that the mantissa isn't normalized though the first bit in the mantissa usually is 1.

This format takes 10 bytes, with (as usual) the most significant bit holding the sign, and the next 15 bits holding the exponent, with a bias of 0x3fff. The remaining 64 bits hold the mantissa. Figure D shows the layout of a temporary real.

Figure D: *The temporary real*



The temporary real can hold numbers with a greater range and precision than either the short or long reals. The mantissa holds about 19 decimal digits of accuracy. The extended exponent allows you to represent numbers in the range from $\pm 1.18 \times 10^{4932}$ to $\pm 3.37 \times 10^{-4932}$. Typically, however, you won't use the temporary real—it's a form the 80*x*87 uses internally.

A couple of examples

Now that we've covered the different number formats, let's look at how to interpret some different values. We'll show you how to disassemble a float, then we'll select a number and convert it to a float.

Example 1—binary to float

As you know, the 80*x*86 processors store the least significant bytes before the most significant bytes, so the number 0B0DC1A09H actually is stored as the four-byte sequence 09H, 1AH, 0DCH, and 0B0H.

If you write the hex number 0B0DC1A09H in binary, you'll get 1011 0000 1101 1100 0001 1010 0000 1001. Therefore, the topmost bit (the sign bit) is 1, the next eight bits are 01100001 (or 61H), and the mantissa is (1)101 1100 0001 1010 0000 1001. (The 1 in parentheses is implied

because it's a normalized mantissa.) Since our bias is 0x7f, we subtract it from 0x61 to get -0x1e or -30. Our sign bit is 1, so our number is negative. The hard part is the mantissa, which is 1*1.0 + 1*0.5 + 0*0.25 + 1*0.125... or 1.719544. The whole number, therefore is $-1 * 1.719544 * 2^{-30}$, or $-1.60145 * 10^{-9}$ (since 2^{-30} is equal to $9.3132257 * 10^{-10}$).

Example 2—float to binary

Now let's convert the number 3.14159 to its binary equivalent. First, we convert the number to pure binary. The part to the left is easy: Just convert it to binary as you would an integer ($3_{10} = 0011_2$). There's an easy way to convert the fractional part to binary as well: Raise 2 to the power of the number of bits in the mantissa, then multiply the fraction by this new number, truncate it, and convert it to binary. This value is the binary fraction.

For example, since we're creating a float, there are 24 bits in the mantissa, so we compute 22^4, which is 16777216. Next, we multiply 0.14159 by 16777216 and get 2375486.013. Truncating this to 2375486 and converting to binary gives us 001001000011111100111110. Now we have converted the decimal number 3.14159 to its binary equivalent: 11.001001000011111100111110.

Now, let's assemble the number. We start our exponent at 0x7f, but we have to normalize our number. We want exactly one binary digit to the left of the point, so we need to shift our mantissa right by one bit. This means that we also must add 1 to the exponent. Then we remove the upper bit from the mantissa (since we know it's 1). The number is positive, so our sign bit is 0.

Now we have 0 for our sign bit, 1000 0000 (80H) for the exponent, and 1001 0010 0001 1111 1001 1111 for the mantissa. When you put the the bits together, you get 0x40490FCF, the hex equivalent of 3.14159.

Special considerations

When you use floating point numbers in your programs, you ought to be aware of one potential source of error. Numbers that you can express exactly in decimal can't necessarily be expressed exactly in binary, and vice versa.

For example, $\frac{1}{3} = 0.3333...$ in decimal, while it's 0.0101010101... in binary. An example of a number expressible exactly in decimal that infinitely repeats in binary is 0.1—the binary representation is 0.100110011001....

As another example, when we converted 3.14159 to binary earlier, we truncated the mantissa, *losing* the fraction 0.013. Therefore, we haven't converted the number 3.14159 to binary, but to the closest binary equivalent available.

Conclusion

Now you know how the 80*x*87 math coprocessor and MASM represent floating point numbers. You'll find this information useful when you start writing functions that work with floating point numbers. You'll also find it useful when you start interfacing assembly language routines with other computer languages, such as C and BASIC. You might also find this information interesting when you start analyzing errors that occur in floating point operations.

READING FOREIGN FILE FORMATS

Translating between GWBASIC and MASM floating point number formats

Listing 3A starts on page 15

Do you have some GWBASIC programs lying around that you still use? Sure! Why rewrite those programs when they work just fine? It would be nice, however, to be able to write new programs that use your data without having to write them in GWBASIC. But GWBASIC stores numbers in a format incompatible with Microsoft Assembler. In this article we'll offer four functions that convert single and double precision GWBASIC numbers to MASM format, and vice versa.

If you're unfamiliar with how MASM stores floating point numbers, you first might want to review the article "How Microsoft Assembler represents floating point numbers," starting on page 6.

Let's kick off this article with a discussion of how the old single-precision floating-point format differs from the new short real floating-point format.

Single precision versus float

Microsoft invented its single-precision format back in the stone age of microcomputing, the 1970s. When Microsoft ported BASIC to the IBM PC, the designers unfortunately

retained the single-precision floating-point format used in CP/M instead of using a format compatible with the IBM PC's coprocessor. (The 80*x*87 family of math coprocessors conforms to the IEEE standard 754.)

Figure A shows the format GWBASIC uses to store single precision numbers in binary data files. Compare this to the format used by the 80*x*87 math coprocessor shown in Figure B.

Figure A: *GWBASIC single-precision format*

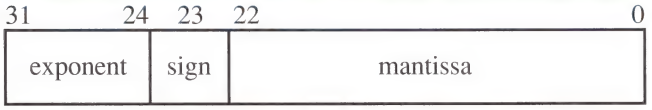
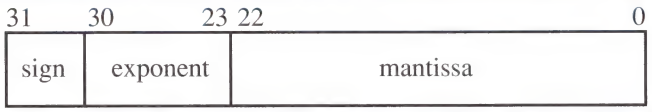


Figure B: *IEEE four-byte real format*



Notice that the two formats are very similar. The signs and exponents trade places, and the exponents have a different bias. Otherwise, the representations are the same. (Actually, there are a few minor differences in the way each standard represents some special values, but those differences have no impact on this discussion.)

The functions `MSBtoIEEEsingle$n` and `IEEEtoMSBsingle$n` perform complementary tasks: The first converts floating point numbers from the old Microsoft format to the IEEE format used by MASM and the 80*x*87, while the second converts IEEE short reals to the old Microsoft format. These two functions, both found in Listing 3A, perform very similar jobs—they swap the sign and the exponent and adjust the exponent's bias value. Both accept pointers to the original number and convert the number in place. Since they're so similar, let's analyze only one of them. We reproduce function `IEEEtoMSBfloat$n` in Figure C.

Before you call `IEEEtoMSBfloat$n`, you must load the DI register with the address of the floating point number to convert to the old Microsoft format. `IEEEtoMSBfloat$n` pushes the AX register onto the stack because you don't want to damage registers unnecessarily. Next, the function reads in the word at DI+2 (the most significant word). As the comment indicates, the word holds the sign bit, eight bits of exponent, and the first seven bits of mantissa.

Next, `IEEEtoMSBfloat$n` shifts AX left for two reasons: it puts the sign bit into the Carry flag (CF) and puts all eight bits of the exponent in a one-byte register (AH). Then the function shifts the AL register right to inject the sign bit into the upper bit. This leaves the upper word in the appropriate format for an old Microsoft float—except for one minor detail.

Figure C: *The function IEEEtoMSBfloat$n*

```
IEEEtoMSBfloat$n proc
    push    ax
    mov     ax, [di+2]  ; AX = SeeeeeeeeMMMMMM
    rcl     ax, 1       ; AX = eeeeeeeeMMMMMM?,  CF=S
    rcr     al, 1       ; AX = eeeeeeeeSMMMMMM
    add     ah, 2       ; AX = EEEEEEEESMMMMMM
    jc      I2Ms_error  ; If exponent overflows -- error
    mov     [di+2], ax  ; Store result
    pop     ax
    clc                 ; Clear carry -- i.e. no errors
    ret
I2Ms_error:             ; Error detected -- don't convert
    pop     ax
    stc                 ; Set carry -- i.e. return error
    ret
IEEEtoMSBfloat$n endp
```

That detail is to readjust the exponent bias. Since IEEE short floats use a bias of 7fH, and the old Microsoft format uses a bias of 81H, the function simply adds 2 to the AH register. If this addition doesn't overflow, `IEEEtoMSBfloat$n` stores the resulting word back into the word pointed to by DI+2, restores the original value of AX from the stack, and clears the Carry flag to tell the caller that all went well.

On the other hand, if the addition does overflow, the function restores the original value of AX and sets the Carry flag—thus informing the caller that it couldn't convert the number to the old Microsoft format.

Since the `MSBtoIEEEfloat$n` function operates so similarly, you should have no trouble figuring out how it works.

Single precision versus double

Sometimes, single precision just isn't precise enough for calculations. For these instances, Microsoft provided the double-precision number format. This format was a simple extension of the single-precision format—Microsoft simply added four bytes of mantissa to the single-precision format. IEEE, however, made both the exponent and mantissa larger to handle a larger range of numbers. Figure D shows the GWBASIC double-precision format, while Figure E shows the IEEE eight-byte format.

Figure D: *GWBASIC double precision*

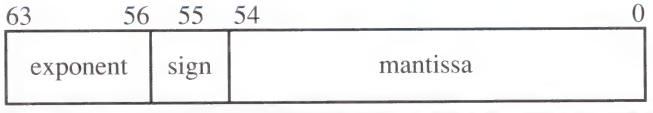
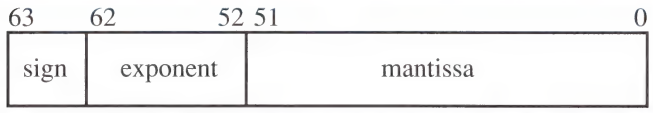


Figure E: *IEEE eight-byte real*



The process of converting Microsoft's old double-precision floating point numbers to IEEE long reals is similar to the process used to convert the single precision numbers to IEEE short reals, with one slight complication. That added difficulty is that you need to shift the mantissa three bits one way or the other because of the size difference of the exponents.

Again, both `IEEEtoMSBdouble$n` and `MSBtoIEEEdouble$n` are so similar that we'll analyze only the first here. Since this function is a bit more complex than `IEEEtoMSBfloat$n`, we'll break our discussion down into three pieces.

In Figure F we present the code for the first part of the `IEEEtoMSBdouble$n` routine—the part that adjusts the bias of the exponent and swaps the exponent and sign bit.

Figure F: *IEEEtoMSBdouble$n routine, part 1*

```
IEEEtoMSBdouble$n proc
    @PushAll
    mov     ax, [di+6]    ; AX = SeeeeeeeeeeeMMMM
    mov     dx, ax        ; <keep a copy...>
    mov     cl, 4
    shr     ax, cl        ; AX = 0000Seeeeeeeeeee
    and     ah, 07h       ; AX = 00000eeeeeeeeeee
    sub     ax, 037eh     ; AX = 00000???EEEEEEEE
    jc      I2Md_error    ; Check exponent range for overflow
    cmp     ax, 100h      ; and underflow
    jnc     I2Md_error
    mov     [di+7], al    ; Store the exponent
    mov     ax, dx        ; AX = SeeeeeeeeeeeMMMM again
```

First the function gets the word containing the sign, exponent, and first four bits of the mantissa, just as occurred in `IEEEtoMSBfloat$n`. It makes a temporary copy of the word by moving it into DX.

Then `IEEEtoMSBfloat$n` extracts the exponent by shifting the AX register right four bits and ANDing off the sign bit. Now the function is ready to adjust the exponent bias. Since an IEEE long double uses a bias of 3FFH and the old Microsoft bias is 81H, the function can adjust the bias by subtracting 37EH. Since the IEEE exponent is 11 bits long and the Microsoft exponent holds only eight bits, underflowing or overflowing the exponent occurs easily. So if the subtraction underflows, the function has found an error.

Next, `IEEEtoMSBfloat$n` checks the result of the subtraction—if it's larger than 0FFH, the exponent would overflow, so again there's an error. After passing these tests, there are no further possibilities for errors. If the exponent passes these tests, the AL register now holds the old Microsoft format exponent, which the function stores into the buffer. Then it restores the copy of the uppermost word to the AX register.

Now we're ready to examine the next part—shifting the mantissa left three bits. We'll do this with the code shown in Figure G.

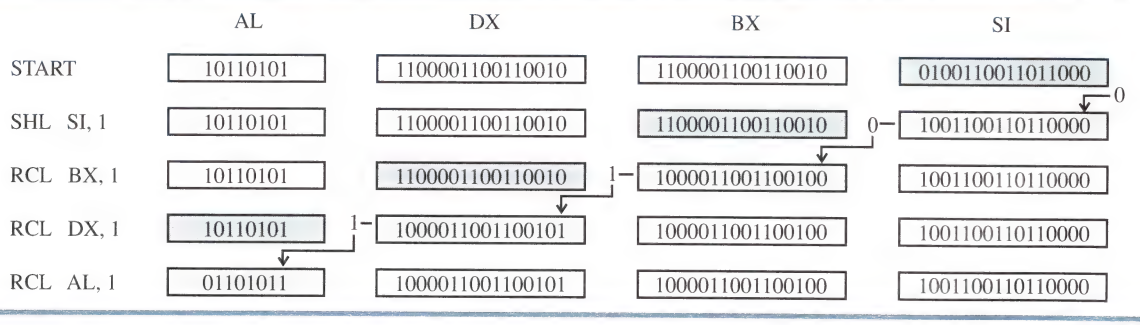
Figure G: *IEEEtoMSBdouble$n routine, part 2*

```
    mov     si, [di]      ; Load next three words
    mov     bx, [di+2]
    mov     dx, [di+4]
    mov     cx, 3         ; # bits to shift
M2Iloop:
    shl     si, 1         ; Shift mantissa left 1 bit
    rcl     bx, 1
    rcl     dx, 1
    rcl     al, 1
    loop    M2Iloop
```

Shifting the mantissa isn't nearly so involved as adjusting the exponent. The function simply loads the rest of the mantissa into the DX, BX, and SI registers. (Remember that the AL register holds the mantissa's first four bits.)

`IEEEtoMSBfloat$n` loads CX with the number of bits to shift, then shifts the entire mantissa to the left, one bit at a time. The `SHL SI, 1` instruction inserts a 0 bit into the lowest bit of the SI register and shifts the highest bit into the CF. Then the `RCL BX, 1` instruction shifts the CF bit into the lowest bit of BX, and the highest bit of BX gets shifted out and winds up in the CF bit. Similarly, the function rotates this new value in the CF into the lowest bit of the DX register, and the upper bit of DX is again the CF. Figure H illustrates how the

Figure H: *Shifting the mantissa left one bit*



SHL and the three RCL instructions work in concert to shift the mantissa left one bit.

When the RCL AL, 1 instruction rotates CF into the lowest bit of AL, it's rotating the highest bit of AL (an exponent bit) into CF, which is of no current concern to us, so we'll ignore it. Then the LOOP instruction executes this entire loop two more times. When the loop is done, the AL register contains the first seven bits of the mantissa and the AH register holds the sign bit.

Now we're ready to move the sign bit to its new location and store the mantissa. The code that performs this is shown in Figure I.

First we need to move the sign bit into its proper place in AL, so IEEEtoMSBfloat$n shifts the entire AX register left. This places the sign bit in CF. (It also shifts a 0 into the lowest bit, but the function will fix that next.) The function rotates CF into the top bit of AL (which shifts that 0 right back out of AL).

Then the function stores the entire mantissa. You'll notice that it stores only the AL register instead of the AX register. (We already stored the exponent in part 1, and right now the AH register contains some messy stuff—certainly not the proper exponent value!)

Figure I: *IEEEtoMSBdouble$n routine, part 3*

```
    shl     ax, 1           ; AX = eeeeeeeeMMMMMM0, CF = S
    rcr     al, 1           ; AX = eeeeeeeeSMMMMMM
    mov     [di], si        ; Store the result
    mov     [di+2], bx
    mov     [di+4], dx
    mov     [di+6], al      ; (already stored exp at DI+7)
    @PopAll
    clc
    ret
I2Md_error:                 ;Error detected--don't convert
    @PopAll
    stc                     ;Set Carry--i.e. error detected
    ret
IEEEtoMSBdouble$n endp
```

Finally, IEEEtoMSBfloat$n restores all the registers to the value they had before you called the function. Now let's examine our demonstration programs, so you can see them work.

Our example programs

We created two programs to demonstrate our functions. The first program, in GWBASIC, simply accepts numbers and prints the equivalent string of hexadecimal bytes. Since this program is so small, we present it as Figure J.

When you run the program, it asks you for a floating point number, and you type one in. Then it presents the string of hex digits that represent your number and prompts you for the next one. You can stop the program by using the ^Break key. Figure K shows some sample output from the program.

Figure J: *The GWBASIC demonstration program*

```
10 ' SHOWFLOT.BAS - show the hexadecimal equivalents of
11 ' floating point numbers in the old Microsoft binary
12 ' format
13 '
20 PRINT "Enter a floating point number: ";
30 INPUT X#
40 P = VARPTR( X# )
50 FOR I=7 TO 0 STEP -1
60    PRINT RIGHT$( "0"+HEX$( PEEK( P+I ) ), 2 ); " ";
70 NEXT I
80 PRINT
90 GOTO 20
99 END
```

Figure K: *Sample output from SHOWFLOT.BAS*

```
Enter a floating point number: ? 1
81 00 00 00 00 00 00 00
Enter a floating point number: ? -75
87 96 00 00 00 00 00 00
Enter a floating point number: ? 3.14159
82 49 0F CF 80 DC 33 72
```

As we mentioned earlier, Microsoft's old double-precision format is a minor extension of the single-precision format—Microsoft simply added four bytes of mantissa. Therefore, when you use SHOWFLOT.BAS, you can just ignore the last four bytes printed if you want to know the hex bytes for single-precision floating point numbers.

The other test program, TESTMSBN.ASM (shown in Listing 3B on page 16), uses an array of short reals and an array of long reals. The program first converts each short real to the old Microsoft format, then back again. It does the same thing for the array of long reals.

We created two macros, PR_STR and PR_STR_HEX, because we use similar code sequences frequently. PR_STR simply prints the string indicated by its argument strofs. PR_STR_HEX also prints a string indicated by the macro argument strofs, then calls prthex—a routine that prints a floating point number as a sequence of hexadecimal digits.

Since the test program uses the same logic for both short and long reals, we'll discuss only the code for the short real conversion. Figure L shows the code for the float conversion loop.

First, TESTMSBN.ASM initializes the short real conversion loop by pointing DI to the first short real in the array, setting CX to the number of reals in the array and SI to the length of a short real.

Next, the program starts a new line and prints the first short real's hexadecimal representation. Then it converts the hex to the old Microsoft single-precision format. If an error occurs, the program prints an error message and skips down to the end of the loop. Otherwise, it prints the hexadecimal representation of the converted number. Then it performs the same steps to convert it back to the IEEE short real format and print its hexadecimal representation.

Figure L: *The start of the float conversion loop*

```
    mov     di, offset shorts ; Point to first short real
    mov     cx, 5             ; Convert 5 short reals
    mov     si, 4             ; Size of short reals
shortLoop:
    PR_STR_HEX newline        ; New line, print IEEE hex
    call    IEEEtoMSBshort$n  ; Convert to MSB format
    jnc     shortSkip1
    PR_STR  Ishort            ; Tell user about cvt error
    jmp     shortSkip3
shortSkip1:
    PR_STR_HEX cvtsto         ; Show what IEEE # cvted to
    call    MSBtoIEEEshort$n  ; convert it back to IEEE
    jnc     shortSkip2
    PR_STR  Mshort            ; Tell user about cvt error
    jmp     shortSkip3
shortSkip2:
    PR_STR_HEX cvtsto         ; Show user it cvts back O.K.
shortSkip3:
    add     di, si            ; point to next float
    loop    shortLoop         ; continue until done
```

At the end of the loop, TESTMSBN.ASM adds the length of the short real (which it stored in SI) so that DI points to the next short real. Then the program jumps to the beginning of the loop to print the next short real. The loop repeats until all five short reals are converted. Figure M shows the TESTMSBN output.

You can modify the arrays of numbers in TESTMSBN.ASM to see what other numbers look like before and after being converted. You'll notice that when you convert between IEEE long reals and the old Microsoft double precision numbers, you may wind up with a difference in the lower three bits because of the differences in the sizes of the mantissas of these different formats.

Figure M: *Output of TESTMSBN*

```
[C:\COBB\IASM\AUGSEP91]msbincvt

3F800000 converts to 81000000 converts to 3F800000
40490FD0 converts to 82490FD0 converts to 40490FD0
402DF84D converts to 822DF84D converts to 402DF84D
45E8EB00 converts to 8D68E800 converts to 45E8EB00
F2813F39 converts to E7813F39 converts to F2813F39
4032000000000000 converts to 8510000000000000 converts to 4032000000000000
4056EAA9930BE0DF converts to 8737554C985F06F8 converts to 4056EAA9930BE0DF
4A5717BA576AA169 Error converting IEEE long real!
C3538A388A43C000 converts to B79C51C4521E0000 converts to C3538A388A43C000
BE17315CDFCE0816 converts to 63B98AE6FE7040B0 converts to BE17315CDFCE0816
[C:\COBB\IASM\AUGSEP91]
```

Conclusion

Now you know the differences between the old Microsoft floating-point formats for single precision and double precision numbers, and the differences between those and the IEEE formats. You can use the routines presented in this article to help you convert data from the old format to the new one.

Source code listings *(You can download the listings from CGIS.)*

The following are the complete programs described in the preceding articles. We placed the listings at the end to preserve continuity, and we restricted them to 76 columns to maintain readability in the two-column format. You can download the listings from our online service, CGIS, at any time of the day. Use your 300-, 1200-, or 2400-baud modem with 8 data bits, 1 stop bit, and no parity to call CGIS at (502) 499-2904. Your user ID is your customer number (the one- to seven-digit number on the top line of your mailing label after the *C:*).

Listing 1: *Examining the Program Segment Prefix (PSP)*

```
;*****************************************************************
;   SHOWPSP.ASM - display the interesting data areas in the Program
;   Segment Prefix data area
;   To assemble: ml /DMEMORY_MODEL=small showpsp.asm hex.asm
;*****************************************************************

    .model  MEMORY_MODEL
    .dosseg
    include STDMAC.INC    ; see colored box on page 16.
    .stack
CRLF    textequ <0dh, 0ah>
ENDLIST textequ <0ffffh>

;-----------------------------------------------------------------
;   printStr - macro to print a string
printStr macro offs:REQ, segreg
    IFNB <segreg>
        push    ds
        mov     dx, segreg
        mov     ds, dx
    ENDIF
    mov     dx, offset offs
    DOSsvc  9
    IFNB <segreg>
        pop     ds
    ENDIF
endm

;-----------------------------------------------------------------
;   printLine - macro to print a string and a numeric value. The
;   value can be a BYTE, WORD, or DWORD
printLine macro string, type, address
    mov     dx, offset string
    DOSsvc  9
    IF type EQ BYTE
        mov al,es:[address]
        call hexout$b
    ELSEIF type EQ WORD
        mov ax,es:[address]
        call hexout$w
    ELSEIF type EQ DWORD
        mov ax,es:[address]
        mov dx,es:[address+2]
        call hexout$d
    ENDIF
endm

    ;-----------------;
    ; Local data area ;
    ;-----------------;
    .data
m$PSPseg    db  CRLF, '        Current PSP segment is - $'
m$segend    db  CRLF, '      Last segment of PSP seg  - $'
m$oldInt22  db  CRLF, '             Terminate address - $'
m$oldInt23  db  CRLF, '          ^Break exit address  - $'
m$oldInt24  db  CRLF, '   Critical error exit address - $'
m$parPSPseg db  CRLF, "                   Parent's PSP - $"
```

```
m$segEnv    db  CRLF, '         Segment of environment - $'
m$INT21stak db  CRLF, ' Stack contents at last INT 21H - $'
m$numFiles  db  CRLF, '          Number of file handles - $'
m$ftabaddr  db  CRLF, '           Address of file table - $'
m$parPSPptr db  CRLF, "         Pointer to parent's PSP - $"
m$FCB1      db  CRLF, '                  Default file 1 - $'
m$FCB2      db  CRLF, '                  Default file 2 - $'
m$CMDlen    db  CRLF, 'Characters in command line tail - $'
m$CMDtail   db  CRLF, 'Command tail: $'
m$fileTbl   db  CRLF, 'File handle table:', CRLF, '  $'
m$BadType   db  'Undefined type specifier'

    ;-----------;
    ; Code area ;
    ;-----------;
    .code
    .startup
    mov     al, '$'             ; String terminator
    mov     es:[68h], al        ; Terminate FCB 1 file name
    mov     es:[78h], al        ; Terminate FCB 2 file name
    mov     bl, es:[80h]        ; Terminate command tail
    xor     bh, bh
    add     bx, 81h
    mov     es:[bx], al

    printStr    m$PSPseg
    mov         ax, es
    call        hexout$w
    printLine   m$segend,       WORD,   02h
    printLine   m$oldInt22,     DWORD,  0ah
    printLine   m$oldInt23,     DWORD,  0eh
    printLine   m$oldInt24,     DWORD,  12h
    printLine   m$parPSPseg,    WORD,   16h
    printLine   m$segEnv,       WORD,   2ch
    printLine   m$INT21stak,    DWORD,  2eh
    printLine   m$numFiles,     WORD,   32h
    printLine   m$ftabaddr,     DWORD,  34H
    printLine   m$parPSPptr,    DWORD,  38H
    printStr    m$FCB1
    printStr    5dh, es
    printStr    m$FCB2
    printStr    6dh, es
    printLine   m$CMDlen,       BYTE,   80H
    printStr    m$CMDtail
    printStr    81H, es

    printStr    m$fileTbl       ; Print the table of file handles
    mov     cx, 20
    les     bx, es:[34h]
@@:
    mov     al, es:[bx]
    call    hexout$b
    mov     dl, ' '
    DOSsvc  2
    inc     bx
    loop    @B
    .exit   0                   ; Exit program

;-----------------------------------------------------------------
;   strprint - print the indicated string
;   INP:    DS:DX - address of string to print
strprint:
    mov ah,09h
    int 21h
    ret

extern  hexout$b:proc, hexout$w:proc, hexout$d:proc
    end
```

Listing 2: *A collection of hexadecimal output functions*

```
;*****************************************************************
;   hex.asm - a library of functions that format and display hexa-
;   decimal numbers.
;*****************************************************************

    title       Inside Assembler library functions
    subtitle    HEX - hexadecimal number formatting and display
    .model  MEMORY_MODEL
    include macros.inc          ; MASM 6.0's macro library
    include stdmac.inc          ; see colored box on page 16.

    ;************;
    ; DATA AREA ;
    ;************;
    .data
hexbuff byte    '????:????$'    ; used by the hexout??? functions as
                                ; a temporary output buffer

    ;************;
    ; CODE AREA ;
    ;************;
    .code

;-----------------------------------------------------------------
;   hexfmt$yn - format a nybble in hexadecimal into the buffer
;   INP:    AL - contains the nybble to format
;           DI - ptr to location to store the formatted character
;   OUT:    DI - incremented to next buffer location
;   USES:   AL
hexfmt$yn proc
    and     al, 0fh             ; Discard the upper nybble
    add     al, '0'             ; Shift range from 0..f to '0'..'?'
    cmp     al, '9'             ; If in range of ':'..'?' shift the
    jng     @F                  ; range to 'A'..'F'
    add     al, 'A'-':'
@@: mov     [di], al
    inc     di
    ret
hexfmt$yn endp

;-----------------------------------------------------------------
;   hexfmt$bn - format a byte in hexadecimal into the buffer
;   INP:    AL - contains the byte to format
;           DI - ptr to buffer to hold the formatted character
;   OUT:    DI - points to the next buffer location
;   USES:   AL
hexfmt$bn proc
    push    ax                  ; Save the lower nybble
    shr     al, 1               ; Move upper nybble to lower
    shr     al, 1
    shr     al, 1
    shr     al, 1
    call    hexfmt$yn           ; Format upper nybble into buffer
    pop     ax                  ; Restore lower nybble
    call    hexfmt$yn           ; Format lower nybble into buffer
    ret
hexfmt$bn endp

;-----------------------------------------------------------------
;   hexfmt$wn - format a word in hexadecimal into the buffer
;   INP:    AX - word to fomat into the buffer
;           DI - ptr to buffer to hold formatted word
;   OUT:    DI - points to next buffer location
;   USES:   AX
hexfmt$wn proc
    xchg    ah, al              ; Swap upper byte with lower byte
    call    hexfmt$bn           ; Format upper byte into buffer
    xchg    ah, al              ;
    call    hexfmt$bn           ; Format lower byte into buffer
    ret
hexfmt$wn endp

;-----------------------------------------------------------------
;   hexfmt$dn - format a double word in hexadecimal into the buffer
;   INP:    DX:AX - the double word to format into the buffer
;           DI - ptr to buffer to hold the formatted double word
;   OUT:    DI - points to next buffer location
;   USES:   AX, DX
hexfmt$dn proc
    xchg    dx, ax              ; Preserve lower word
    call    hexfmt$wn           ; Format upper word into buffer
    mov     byte ptr [di], ':'  ; Insert ':' delimiter into buffer
    inc     di
    xchg    dx, ax              ; Restore lower word
    call    hexfmt$wn           ; Format lower word into buffer
    ret
hexfmt$dn endp

;-----------------------------------------------------------------
;   hexout$y - output a hexadecimal nybble to the screen
;   INP:    AL - the lower half is the nybble to output
;   USES:   AL
hexout$y proc
    @SaveRegs di, dx
    mov     dx, offset hexbuff  ; Start of hex output buffer
    mov     di, dx              ; Set pointer for hexfmt
    call    hexfmt$yn           ; Format nybble into buffer
    mov     byte ptr [di],'$'   ; Terminate the string
    DOSsvc  9                   ; Print the string
    @RestoreRegs
    ret
hexout$y endp

;-----------------------------------------------------------------
;   hexout$b - output a hexadecimal byte to the screen
;   INP:    AL - the byte to output
;   USES:   AX
hexout$b proc
    @SaveRegs   di, dx
    mov     dx, offset hexbuff  ; Start of hex output buffer
    mov     di, dx              ; Set pointer for hexfmt
    call    hexfmt$bn           ; Format byte into buffer
    mov     byte ptr [di],'$'   ; Terminate the string
    DOSsvc  9                   ; Print the string
```

```
    @RestoreRegs
    ret
hexout$b endp

;--------------------------------------------------------------------
;   hexout$w - output a hexadecimal word to the screen
;   INP:    AX - the word to print
;   USES:   AX
hexout$w proc
    @SaveRegs   di, dx
    mov     dx,offset hexbuff   ; Start of hex output buffer
    mov     di, dx              ; Set pointer for hexfmt
    call    hexfmt$wn           ; Format the word into the buffer
    mov     byte ptr [di],'$'   ; Terminate the string
    DOSsvc  9                   ; Print the string
    @RestoreRegs
    ret
hexout$w endp

;--------------------------------------------------------------------
;   hexout$d - print a double word in hexadecimal on the screen
;   INP:    DX:AX - the double word to print
;   USES:   AX, DX
hexout$d proc
    @SaveRegs   di, dx, cx
    mov     cx, offset hexbuff  ; Start of hex output buffer
    mov     di, cx              ; Pointer value for hexfmt
    call    hexfmt$dn           ; Format double word into the buffer
    mov     byte ptr [di],'$'   ; Terminate the string
    mov     dx, cx              ; <we stored str in CX because
                                ;  hexfmt$dn destroys DX>
    DOSsvc  9                   ; Print the string
    @RestoreRegs
    ret
hexout$d endp

    end
```

Listing 3A: *Translating between GWBASIC and MASM*

```
;*********************************************************************
;
;   MSBINCVT.ASM - subroutines to convert between the (obsolete)
;   Microsoft Binary Format and IEEE format numbers.
;   IMPORTANT NOTE: All these functions manipulate the relative
;   positions of the sign bit and exponent fields. Therefore, the
;   comments will often look like:
;
;   AX = EEEEEEEESMMMMMM
;   Where we use:
;       M = Mantissa bit
;       E = Microsoft exponent bit
;       S = Sign bit
;       e = IEEE exponent bit
;*********************************************************************

.model MEMORY_MODEL
    include MACROS.INC          ; @PopAll,@PushAll
.code

;--------------------------------------------------------------------
;   MSBtoIEEEshort$n - convert an old Microsoft single precision
;   float to IEEE short real format
;   INP:    DI - points to the MSB number
;   OUT:    CY - TRUE if exponent range error detected, FALSE o/w
;           the number is converted, in place, to IEEE format
;   USES:   flags
MSBtoIEEEshort$n proc
    push    ax
    mov     ax, [di+2]          ; AX = EEEEEEEESMMMMMM
    sub     ah, 2               ; AX = eeeeeeeeSMMMMMM fix exponent
    jna     M2Is_error          ; If exponent out of range, error
    rcl     al, 1               ; AX = eeeeeeeeMMMMMM?,  CF = S
    rcr     ax, 1               ; AX = SeeeeeeeeMMMMMM
    mov     [di+2], ax          ; Store result
    pop     ax
    clc                         ; Clear carry -- i.e. no errors
    ret
M2Is_error:                     ; Error detected -- don't convert
    pop     ax
    stc                         ; Set carry -- i.e. return error
    ret
MSBtoIEEEshort$n endp

;--------------------------------------------------------------------
;   IEEEtoMSBfloat$n - convert an IEEE short real to old Microsoft
;   single precision format
;   INP:    DI - points to the IEEE number to convert
;   OUT:    CY - TRUE if exponent range error detected, FALSE o/w
;           the number is converted, in place, to the old MS format
;   USES:   flags
IEEEtoMSBshort$n proc
    push    ax
    mov     ax, [di+2]          ; AX = SeeeeeeeeMMMMMM
    rcl     ax, 1               ; AX = eeeeeeeeMMMMMM?,  CF=S
    rcr     al, 1               ; AX = eeeeeeeeSMMMMMM
    add     ah, 2               ; AX = EEEEEEEESMMMMMM
    jc      I2Ms_error          ; If exponent overflows - error
    mov     [di+2], ax          ; Store result
    pop     ax
    clc                         ; Clear carry -- i.e. no errors
    ret
I2Ms_error:                     ; Error detected -- don't convert
    pop     ax
    stc                         ; Set carry -- i.e. return error
    ret
IEEEtoMSBshort$n endp

;--------------------------------------------------------------------
;   MSBtoIEEElong$n - convert Microsoft Binary long precision to
;   IEEE long real format
;   INP:    DI - points to the MSB number
;   OUT:    CY - FALSE (for consistency-no errors are possible)
;           the number is converted, in place, to IEEE format
;   USES:   flags
;   NOTE:   No error possible (values ALWAYS fit)
MSBtoIEEElong$n proc
    @PushAll
    ; Shift entire 64 bit number right 3 bits
    ; At start: AX/DX/DI/SI = EEEEEEEESMMMMMMMMMMMMMMMM.....
    mov     si, [di]            ; Load AX:DX:BX:SI
    mov     bx, [di+2]
    mov     dx, [di+4]
    mov     ax, [di+6]
    mov     cx, 3               ; # bits to shift
M2Iloop:
    shr     al, 1               ; Shift mantissa one bit right
    rcr     dx, 1
    rcr     bx, 1
    rcr     si, 1
    loop    M2Iloop
    mov     [di], si            ; Store lower three words
    mov     [di+2], bx
    mov     [di+4], dx
    ; Now AX = EEEEEEEE000SMMMM

    ; Adjust exponent & swap around sign & exponent fields
    mov     dl, al              ; DL = 000SMMMM
    xchg    al, ah
    xor     ah, ah              ; AX = 00000000EEEEEEEE
    add     ax, 037eh           ; AX = 00000eeeeeeeeeee
    test    dl, 10h             ; Check S, if on set new S in AX
    jz      M2Iskip
    or      ah, 08h
M2Iskip:                        ; AX = 0000Seeeeeeeeeee
    mov     cl, 4
    shl     ax, cl              ; AX = Seeeeeeeeeee0000
    and     dl, 0fh             ; DL = 0000MMMM
    or      al, dl              ; AX = SeeeeeeeeeeeMMMM
    mov     [di+6], ax          ; Store the new upper word
    @PopAll
    clc                         ; clear carry -- i.e. no errors
    ret
MSBtoIEEElong$n endp
;--------------------------------------------------------------------
;   IEEEtoMSBdouble$n - convert IEEE double to Old Microsoft format
;   INP:    DI - points to the MSB number
;   OUT:    CY - TRUE if exponent range error detected, FALSE o/w
;           the number is converted, in place, to Old Microsoft format
;   USES:   flags
;   NOTE:   No error checking performed
IEEEtoMSBlong$n proc
    @PushAll
    ; Adjust exponent & swap around sign & exponent fields
    mov     ax, [di+6]          ; AX = SeeeeeeeeeeeMMMM
    mov     dx, ax              ; <keep a copy...>
    mov     cl, 4
    shr     ax, cl              ; AX = 0000Seeeeeeeeeee
    and     ah, 07h             ; AX = 00000eeeeeeeeeee
    sub     ax, 037eh           ; AX = 00000???EEEEEEEE
    jc      I2Ml_error          ; Check exponent range for overflow
    cmp     ax, 100h            ; and underflow
    jnc     I2Ml_error
    mov     [di+7], al          ; Store the exponent
    mov     ax, dx              ; AX = SeeeeeeeeeeeMMMM again

    ; Shift entire 64 bit number left 3 bits
    ; At start: AX/DX/DI/SI = 000SeeeeeeeeMMMMMMMMMMMMMMMM.....
    mov     si, [di]            ; Load next three words-DX:BX:SI
    mov     bx, [di+2]
    mov     dx, [di+4]
    mov     cx, 3               ; # bits to shift
```

```
M2Iloop:
    shl     si, 1               ; Shift mantissa left 1 bit
    rcl     bx, 1
    rcl     dx, 1
    rcl     al, 1
    loop    M2Iloop
    ; Now AX = SeeeeeeeeMMMMMM
    shl     ax, 1               ; AX = eeeeeeeeMMMMMM0, CF = S
    rcr     al, 1               ; AX = eeeeeeeeSMMMMMM
    mov     [di], si            ; Store the result
    mov     [di+2], bx
    mov     [di+4], dx
    mov     [di+6], al          ; (already stored exp at DI+7)
    @PopAll
    clc
    ret
I2Ml_error:                     ; Error detected - don't convert
    @PopAll
    stc                         ; Set carry - i.e. error detected
    ret
IEEEtoMSBlong$n endp

    end
```

Listing 3B: *Translating between GWBASIC and MASM*

```
;*************************************************************************
;   TESTMSBN.ASM - test the floating point format conversion routines
;   To assemble, type the following line:
;   ml /DMEMORY_MODEL=small testmsbn.asm msbincvt.asm hex.asm
;*************************************************************************

CRLF    TEXTEQU <0dh, 0ah>
    .model MEMORY_MODEL
    .dosseg
    include MACROS.INC
    include STDMAC.INC  ; see colored box on page 16.

PR_STR macro strofs:REQ
    mov     dx, offset strofs
    DOSsvc  9
endm

PR_STR_HEX macro strofs:REQ
    PR_STR  strofs
    call    prthex
endm

    .stack
;-------------------------------------------------------------------
;   Here is a list of short and long reals that we'll convert from
;   IEEE format to Old Microsoft format, and then back again. You can
;   view them in CodeView ---- see the article text
.data
shorts  dd      1.0000, 3.14159, 2.71828, 7.453e3, -5.12e30
longs   dq      18.000, 91.6666, 1.35e50, -2.2e16, -1.35e-9
Ishort  byte    " Error converting IEEE short real!$"
Mshort  byte    " Error converting Microsoft single!$"
Ilong   byte    " Error converting IEEE long real!$"
Mlong   byte    " Error converting Microsoft double!$"
cvtsto  byte    " converts to $"
newline byte    CRLF, "$"

;-------------------------------------------------------------------
.code
    extern IEEEtoMSBshort$n:proc, IEEEtoMSBlong$n:proc
    extern MSBtoIEEEshort$n:proc, MSBtoIEEElong$n:proc
    extern hexout$b:proc
.startup

    ;---------------------------------;
    ; Print the single precision floats ;
    ;---------------------------------;
    mov     di, offset shorts   ; Point DI to the first short real
    mov     cx, 5               ; Convert 5 short reals
    mov     si, 4               ; Tell prthex we're using short reals
shortLoop:
    PR_STR_HEX newline          ; Start new line, print IEEE hex
    call    IEEEtoMSBshort$n    ; Convert to MSB format
    jnc     shortSkip1
    PR_STR  Ishort              ; Tell user about conversion error
    jmp     shortSkip3
shortSkip1:
    PR_STR_HEX cvtsto           ; Show user what IEEE # converted to
    call    MSBtoIEEEshort$n    ; Convert it back to IEEE
    jnc     shortSkip2
    PR_STR  Mshort              ; Tell user about conversion error
    jmp     shortSkip3
shortSkip2:
    PR_STR_HEX cvtsto           ; Show user it converts back O.K.
shortSkip3:
    add     di, si              ; point to the next float
    loop    shortLoop           ; continue until we've done them all

    ;-------------------------;
    ; Convert the long reals ;
    ;-------------------------;
    ; Since array of long reals starts immediately after floats, DI
    ; already points to the first long real
    mov     cx, 5               ; Convert 5 long reals
    mov     si, 8               ; Tell prthex we're using long reals
longLoop:
    PR_STR_HEX newline          ; Start new line, print IEEE hex
    call    IEEEtoMSBlong$n     ; Convert to MSB format
    jnc     longSkip1
    PR_STR  Ilong               ; Tell user about conversion error
    jmp     longSkip3
longSkip1:
    PR_STR_HEX cvtsto           ; Show user what IEEE # converted to
    call    MSBtoIEEElong$n     ; Convert back to IEEE
    jnc     longSkip2
    PR_STR  Mlong               ; Tell user about conversion error
    jmp     longSkip3
longSkip2:
    PR_STR_HEX cvtsto           ; Show user it converts back O.K.
longSkip3:
    add     di, si              ; Point to the next double
    loop    longLoop            ; continue until we've done them all

    .exit   0

;-------------------------------------------------------------------
;   prthex - print a sequence of bytes in hexadecimal, separated by
;   spaces
;   INP:    DI - pointer to the bytes to print
;           SI - number of bytes to print
;   USES:   AX, DX
prthex:
    @SaveRegs   cx, di
    mov     cx, si              ; # bytes to print in hex
    add     di, cx              ; point to the last byte
errorLoop:
    dec     di                  ; Adjust counter
    mov     al, [di]            ; get current byte
    call    hexout$b            ; print the byte in hex
    loop    errorLoop           ; continue until all bytes printed
    @RestoreRegs
    ret

    end
```

This listing is used in *all* programs in this issue.

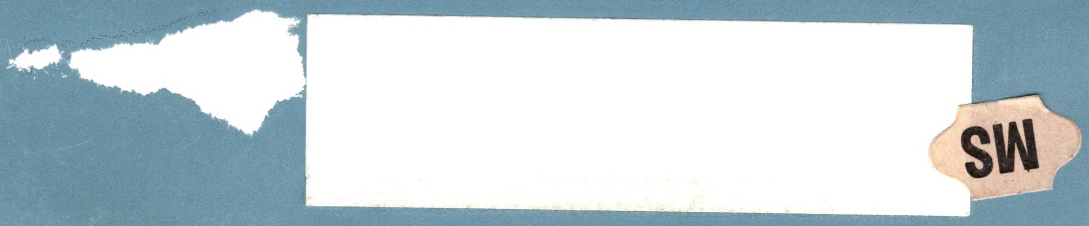
```
;*************************************************************************
;   STDMAC.INC - A collection of macros that we'll use frequently
;   in the Inside Assembler journal. As required, we'll enhance it.
;*************************************************************************

;-------------------------------------------------------------------
;   DOSsvc - macro that performs a DOS service call.
DOSsvc macro service
        mov     ah, service
        int     21h
endm
```

UPCOMING

Soon You'll Be Seeing These Topics...

- Save space or add speed to programs written in Basic, Pascal, or C
- Create bug-free, memory-resident programs
- Getting the most out of macros
- Using DOS and BIOS services
- Defining structures in Microsoft Assembler
- Using data structures
- Accessing EMS and XMS memory
- How to write a device driver
- Interrupt-driven serial communications
- Controlling the parallel port
- Tips for using LIB, LINK, NMAKE, ...
- Strategies for optimizing your code



COBB

INSIDE ASSEMBLER™

Tips & techniques for 80x86 Assembler